Trustworthy Composition: The System is Not Always the Sum of Its Parts

Robert J. Ellison, Software Engineering Institute [vita[1]]

2005-09-28

This document surveys several of the profound technical problems and challenges faced by practitioners in the assembly and integration of secure and survivable systems.

The design of trustworthy networked information systems presents profound challenges for system architecture and project planning. While there has been significant research on this topic, there is still limited understanding of the issues. The software development tools, infrastructure components, and standardized interfaces now available encourage the deployment of networked systems and make that lack of understanding an increasing liability.

The objective of this article [2] is to summarize the technical issues associated with the assembly of a networked information system that confront the practitioner. It is critical that the practitioner understands the limitations of current techniques and hence maintains a healthy skepticism about the assurance associated with a complex software-intensive system, as well as for any “silver bullets” proposed to mitigate that complexity. Integration techniques that make it easier to customize a deployed system can generate a new class of composition errors. The expanding scope of connectivity for a networked information system in terms of devices, users, and organizations often invalidates the design assumptions that simplified the development of today’s legacy systems, but those assumptions may still be guiding new development.

The Committee on Information Systems Trustworthiness was convened by the Computer Science and Telecommunications Board (CSTB) of the National Research Council (NRC) to assess the nature of information systems trustworthiness and the prospects for technology that would increase it. Their report was issued as the document *Trust in Cyberspace* [Schneider 99]. Their report is an excellent summary of the issues and the research required to address them. (Their NIS acronym refers to Networked Information Systems.)

> System-level trustworthiness requirements are typically first characterized informally. The transformation of these informal notions into precise requirements that can be imposed on individual system components is difficult and often beyond the current state of the art. Whereas a large software system such as an NIS cannot be developed defect-free, it is possible to improve the trustworthiness of such a system by anticipating and targeting vulnerabilities. But to determine, analyze, and, most importantly, prioritize these vulnerabilities, a good understanding is required for how subsystems interact with each other and with the other elements of the larger system -- obtaining such an understanding is not possible today.
>
> NISs pose new challenges for integration because of their distributed nature and the uncontrollability of most large networks. Thus, testing subsets of a system cannot adequately establish confidence in an entire NIS, especially when some of the subsystems are uncontrollable or unobservable as is likely in an NIS that has evolved to encompass legacy software. In addition, NISs are generally developed and deployed incrementally. Techniques to compose subsystems in ways that contribute directly to trustworthiness are, therefore, needed.

[1] daisy:208 (Ellison, Robert J.)

[2] This document is derived from [Ellison 05].

The problems are not so much with the assembly and integration process but with properties and interaction of the components. Unfortunately the *Trust in Cyberspace* report also noted why it is difficult to build a knowledge base of best practices to resolve these difficulties.

> There exists a widening gap between the needs of software practitioners and our ability to evaluate software technologies for developing moderate- to large-scale systems. The expense of building such systems renders infeasible the traditional form of controlled scientific experiment, where the same system is built repeatedly under controlled conditions but using differing approaches. Benefits and costs must be documented, risks enumerated and assessed, and necessary enhancements or modifications identified and carried out. One might, instead, attempt to generalize from the experiences gained in different projects. But to do so and reach a sound conclusion requires understanding what aspects of a system interact with the technology under investigation. Some advantages would probably accrue if only software developers documented their practices and experiences. This activity, however, is one that few programmers find appealing and few managers have the resources to support.

The gap between practitioner needs and evaluation techniques is wider now than in 1999. The deployment of networked information systems has accelerated by the emergence of market-driven protocols, such as those associated with Web Services, but without significant progress on how to analyze those systems.

This report is one of the longer BSI documents. The problem as described in the *Trust in Cyberspace* report is complex and multidimensional. This note suggests that considering the composition and integration issues in terms of what can go wrong can be an effective way to reduce the scope of the overall problem. Four classes of errors are described, and any one of those four sections can be read independently of the others.

System Decomposition

The composition problem is of our own creation. We build a large system by first decomposing it into pieces that are more easily managed. Such decompositions might be constrained by a need to integrate legacy systems, a requirement to use a commercially available component, or a desire to reduce costs by reusing an available component. The challenge is to decompose the system in such a way that those individual pieces can be individually implemented and that the composition of those components meets system requirements. Principles such as “Economy of Mechanism” and “Least Common Mechanism” proposed by Saltzer and Schroeder and “Constrained Dependency” proposed by Neumann are examples of decomposition guidelines [Saltzer 75, Neumann 04].

> Decomposition into smaller pieces is a fundamental approach to mastering complexity. The trick is to decompose a system in such a way that the globally important decisions can be made at the abstract level, and the pieces can be implemented separately with confidence that they will collectively achieve the intended result. (Much of the art of system design is captured by the bumper sticker “Think globally, act locally.”) Jim Horning [Neumann 1995]

Of course, we would like assembly and integration of the final system from those pieces to be like snapping Lego blocks together. Unfortunately our final assembly for a large system might not be that neat and may employ ad hoc techniques to integrate some pieces. Components may have multiple personalities. A component that is well behaved when used in one context may generate serious problems when used in another. While individual components might individually meet security requirements, a system composed from those components is not necessarily secure.

Not only do we have to build a system that meets the functional and non-functional requirements, but we also have to build it in such a way that we can demonstrate that those requirements are satisfied. Those requirements have to continue to be satisfied as the system evolves with respect to hardware, networks, and operations. Since such changes over the life of a system can be quite dramatic, providing the evidence that supports assurance is not a one-time event but an activity that continues over the life of the system.

Business Requirements

There are increasing demands for integration and interoperability of business systems. The multiple computing platforms and the lack of widely accepted standards complicate meeting such requirements within most corporations. The World Wide Web has demonstrated a successful approach to sharing information among heterogeneous systems, and that success has increased the demand for integration and interoperability across organizational boundaries. The combination of the Extensible Markup Language (XML) and Web Services represents one technical approach that supports distributed integration and platform interoperability.

Any discussion of best practices for assembly and integration is complicated by the ever-expanding scope. We could consider integration of components within a single executable program, the integration of separate processes on a single host, or the integration of separate processes executing on different hosts. The coupling among components might be very tight for a system controlling a manufacturing process, whereas only limited integration and interoperability may be required between a manufacturing control system and the business systems responsible for sales or inventory. Integration is more difficult as business processes extend beyond corporate boundaries and the remote services are independently developed and managed.

Business integration and assembly requirements are closely coupled with internal and external business processes, and those processes are rarely static. A requirement for better integration of business processes may be coupled with a requirement for more rapid deployment of the computing systems, with an objective for just-in-time deployment. The automation provided by some application frameworks speeds application development, but those frameworks can generate applications faster than they can be assured. While a number of technologies may enable just-in-time software deployment, there has not been the corresponding progress in the ability to provide just-in-time assurance. Rapid deployment and a desire for lower costs often lead to dealing with components that do not quite “fit,” e.g., existing components, commercial off-the-shelf (COTS) tools, and legacy systems.

Business Non-Functional Requirements

An analysis of assembly and integration issues often concentrates on the non-functional requirements. Grady Booch, a co-founder of Rational Systems and now an IBM fellow, made the following comment on March 22, 2005 in his Web log [Booch 05]:

> Most enterprise systems are architecturally very simple yet quite complex in manifestation: simple because most of the relevant architectural patterns have been refined over decades of use in many tens of thousands of systems and then codified in middleware; complex because of the plethora of details regarding vocabulary, rules, and non-functional requirements such as performance and security. Systems of other genres (such as artificial intelligence) are often far more complex architecturally.

For example, the functional architecture required for a manager to review employee information from an HR database is a relatively simple query and display. The use of access rules to support an internal corporate policy is straightforward. When such employee information moves among internal corporate systems or is exported to other organizations such as an insurance provider, the access and usage policy that was straightforward to implement in a single application must now be maintained across multiple applications and organizations. Exporting data also implies exporting the access policy that must be enforced. A legacy system might have implemented a static access policy by implementing that policy in application coding. On the other hand, reuse of a component in multiple contexts with differing access policies might lead to an implementation where the data provides a link to the policy represented in a manner that can be interpreted by the component. A simple binding of access decisions at compile time for the legacy system has been replaced by a more complex dynamic binding of the access policy at runtime.

What Can Go Wrong

There are several systematic, in-depth, and lengthy discussions of the difficulties associated with building trustworthy systems [Anderson 01, Neumann 04, Schneider 99]. The 391 references in the Neumann report are a historical record of the research in this domain. Neither the Neumann nor the Schneider work is organized to meet the needs of the practitioner.

The practitioner should be aware of the ever-lengthening list of problems associated with the composition of software components, but even an expert can find it difficult to navigate through a full collection of the known technical errors. This report suggests an approach for handling composition errors that may help developers identify underlying technical problems and viable resolution strategies.

This report considers software composition from the perspective of four software artifacts. The artifacts represent four views of a system at different levels of abstraction. The chosen artifacts also provide a way to discuss the impact of distributed information systems on the required software assurance analysis and how commercial development trends such as service-oriented architectures (SOA) and Web Services may mitigate some problems but exacerbate others. The four artifacts are

1. **Specific interface**. An interface controls access to a service. Interfaces that fail to validate the input are frequent members of published vulnerability lists.

2. **Component-specific integration**. Assembly problems often arise because of conflicts in the design assumptions for the components. Project constraints may require using components, COTS software, or legacy systems that were not designed for the operating environment, which raises the likelihood of mismatches. The increasing importance of business integration requirements compounds the component integration problems and is the motivation for designs based on SOA.

3. **Architecture integration mechanisms**. Commercial software tool vendors often provide the capability for the purchaser to integrate the tool into their systems and tailor its functionality for their specific needs. However, the capability to reconfigure a system rapidly is matched by the increased probability of component inconsistencies generated by the more frequently changing component base, as well as the increased risk that the dynamic integration mechanisms could be misused or exploited. These mechanisms represent another interface that must be properly constrained.

4. **System behavior: component interactions**. The behavior of a system is not the simple sum of the behavior of the individual components. System behavior is strongly influenced by the interactions of its components. Components may individually meet all specifications, but when they are composed into a system the unanticipated feedback among components can lead to unacceptable system behavior. Security and safety are system rather than component requirements. We can build a reliable system out of unreliable components by appropriate use of redundancy. Components that are not secure as standalone components in an operating environment may be secure when used within the constraints maintained by a system.

As the networked information systems extend beyond corporate boundaries, fewer assumptions can be made about the quality of the accessed services. The size of a system may not be as pressing a factor as the multiplicity of connections, each with unique characteristics. There will be variances in protocols, networks, platforms, the value of the resource, and the confidence in the provider. A key system assumption should be that errors are a normal rather than an exceptional event. Error recovery and the propagation of error conditions among components are among the component interactions that can significantly influence global system behavior.

I. Specific Interface

An interface can be considered a contract between the caller and provider of a service. The service requester has an obligation to provide the necessary input that satisfies the required service preconditions. The successful completion depends on satisfying postconditions [Meyer 92]. Vulnerabilities such as buffer overflows are examples where the preconditions of an interface are not satisfied.

The analysis of an interface involves

- establishing the preconditions and postconditions [The identification of the preconditions often requires analyzing how a service could be misused.]
- analyzing the impact of an interface failure on both the caller and service provider

Frequently mentioned errors such as a buffer overflow are associated with simple interfaces. The increasing use of more complex interfaces increases the importance of this class of composition problems. Networked systems often replace a simple, statically defined, procedural interface with a document-based interface where the content is interpreted by the remote service (e.g., HTML, JavaScript, SQL commands, a Web Service XML-data stream, or a command expressed in a scripting language). That document may represent a well-vetted contractual agreement between the requester and provider, or it may capture a one-time dynamically negotiated agreement. An exploit or an inadvertent error that modifies such a data stream can significantly change the semantics of the request.

The interfaces that provide access to externally developed software components and multifunction tools deserve special attention. While the marketplace for COTS components such as Web server, database, or network or system administration tools seems to require ever-expanding functionality for those tools, the specific usage only requires a subset of the available functionality. A poorly constrained interface might be exploited to access other services. Proper integration should include the configuring of such systems to control usage, as well as establishing the necessary constraints in the interface to restrict access to functionality.

II. Component-Specific Integration

An interface represents the advertised interactions among components. Component integration has to resolve the unanticipated problems created by conflicts among design or implementation decisions made for individual components.

The objective to have software assembly correspond to mechanical assembly goes back to the beginnings of Software Engineering. Doug McIlroy at the 1968 NATO Conference on Software Engineering expressed that sentiment in a talk on "'Mass Produced' Software Components" [McIlroy 68].

> Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and timespace performance. Existing sources of components - manufacturers, software houses, users' groups and algorithm collections - lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors.

The reality associated with component assembly is captured by Garlan et al. [Garlan 94]. They attempted to integrate four mature components:

- a public domain object-oriented database
- a tool for constructing graphical user interfaces
- a commercially available event-broadcast mechanism
- a Remote-Procedure-Call Interface Generator

Although all components were written in C or C++, all were widely used in projects, and all had available source code, a projected schedule of six months and one person-year of effort turned into two years and nearly five person-years of effort to generate a very large and sluggish prototype that was difficult to maintain. While multiple components used an event loop for control, those event loops were not compatible. There were similar conflicts in the assumptions on how components managed data.

The authors referred to these integration problems as architectural mismatches. Architectural mismatches arise from conflicts generated by differing assumptions components make about the structure of the system. The assumptions are usually implicit, which makes them very difficult to analyze in advance of building the systems.

The Garlan paper appeared in 1994. In 2005, a contributor to Neumann's *Risks Digest* posted a statement similar to McIlroy's position on the need for easy assembly of software components [Risks Digest 05]. The replies to that post capture many of the current issues with respect to software assembly. These are some of the comments made in those replies.

1. When components are used in new situations, any existing assumptions cannot be relied on at all, without tedious and careful work to reestablish them.
2. The RISK? Not looking deeply enough to find all the RISKS.
3. The problem isn't a lack of components; it's that we're building much larger systems in relation to the power of those components.
4. COTS - In one interesting way, the problem with present software is not lack of components but the use of them without adaptation. One size fits all is the norm for software and the standard executable is given to millions of users, completely ignoring any variances in knowledge, ability, interest, or configuration.

   Note: The first four items raise the problem of subtle mismatches and the difficulty raised by Garlan [Garlan 94] and others in identifying those. As the size of the system increases, so does the number of components and interfaces. That complexity is a major roadbock for analysis.
5. For a critical real-time application, you have to know the response time. The black box philosophy of Object Oriented design does not work here.
6. Software components are not physical components. They do not scale the same way.

   Note: Items five and six both observe that integration problems occur with the non-functional behavior of a system. An assembled set of components may generate an unexpected pattern of usage of a shared resource, or collective dependencies may result in deadlock or infinite loops as the system scales up in size.
7. We desire to reuse software. And because of the perception that software can do anything, the requirements tend to be complex too: arguably excessively so. This tends to mean that the requirements for each system are unique. Working this down into the details of implementation, this means that the components needed tend to be unique for each system—thus limiting the possibilities of reuse.

8. Of course we reuse code. We reuse entire applications all the time—nobody writes their own Web server when they can simply install Apache.

   Note: Lampson considered the profitability of selling components, and his conclusion that only large-scale reuse would survive is demonstrated by response eight [Lampson 03].

There are significant technical issues for component integration. Neumann has an extensive bibliography of the research for just trustworthy composition [Neumann 04]. Although many of the integration problems remain unsolved, emerging business requirements for integrating networked information systems are generating new ones. The demand is ahead of the development of the foundations needed to adequately support integration. SOAs that have been proposed to address some of the business needs will raise additional integration issues.

Technical Issues

Neumann discusses the factors that complicate composing components into a system [Neumann 04]. Systems do not necessarily have requirements that enable the integration of components. Critical items, such as the state shared by components, may not be adequately specified. While the specifications describe what a software component should do, they often do not specify what it should not do. There will be properties that manifest themselves only because of a combination of subsystems or that appear only after a system has been scaled to a larger capacity or scope.

The mismatches for independently developed components may arise from differing requirements for non-functional system attributes such as performance, reliability, and security or often simply from the paucity of such requirements. Specifying the desired collective behavior for a component involves tradeoffs among those properties. A mechanism that improves modifiability might reduce performance. For example, reliability analysis depends on specific usage to identity the most critical errors and the kind of recovery required for that context. In some situations, the recovery time will be critical, while in others the cost factors may lead to simple restart procedures. While “graceful shutdown” is a popular requirement, creating a graceful shutdown for an assembled system requires an understanding of how each component’s implementation of its concept of a graceful shutdown affects the shutdown of the components that use the provided service. Security will have to deal with similar variances in the tradeoffs made.

There will be mismatches among components that simply cannot be resolved. In practice, a small number of non-functional requirements, often including performance, are the primary drivers for the software architecture. If the mismatch involves one of the primary drivers, it may not be possible to integrate the component. A primary driver for a business application could be a requirement to support geographically distributed business processes with reusable components. Such a requirement might lead to a decision to use a loosely coupled application architecture such as an SOA, along with Web Services. The XML data stream associated with that approach may create a network or computational load that is incompatible with systems with hard real-time requirements.

The evaluation of an existing component for use in a system involves at least

- establishing a level of assurance that the component satisfies its published specifications
- verifying that the component satisfies the architecture specifications
- analyzing the component design and implementation to identify potential architectural mismatches as well as the impact on other requirements, such as operations and system administration
- reviewing and, if necessary, revising the existing risk analysis for any new threats
- analyzing potential revisions to the architecture to resolve any unsatisfied requirements or to mitigate risks associated with usage

The Business Drivers for Component Integration: Service-Oriented Architecture—A Domain-Specific Approach

The discussion so far has concentrated on some of the technical problems associated with assembly and integration. It is simply a very difficult problem with a limited theoretical base on which to build best practices. Neumann stresses the importance of a disciplined approach to all aspects of the development process [Neumann 04]. Multiple lists of design principles and guides to security design patterns have been published [Blakley 04]. (See Principles content.) There have been earlier efforts such as Object Management Group’s Common Object Request Broker Architecture (CORBA), which is designed to support integration across large distributed systems.

We are now in the midst of another attempt to support reuse and improve assembly and integration, this time exploiting the advantages of SOA and Web Services. Compared to earlier attempts, there is now a business demand that could generate a large market, as well as a design paradigm that reflects the lessons learned from the earlier attempts and from the success of the Web. This combination even with only limited success will certainly affect how we build the next generation of systems. The combination of factors that led to this state includes the following:

- The Web architecture demonstrated an approach for easy information interchange across diverse computing platforms. The protocol was stateless.
- The impact of the Web on business has led to a need for better interoperability across organizational boundaries.
- Internal business processes often involve multiple computing platforms with differing security policies, as financial, administrative, and manufacturing control systems are more closely coupled.
- Continued pressure on IT expenses motivates the desire to use and easily tailor a computer-supported service for multiple business processes in order to lower maintenance costs as the business processes evolve. The adjective *agile* is frequently applied, and the “IT bottleneck” is a popular target for complaints.

There are multiple definitions of SOA. This note considers it as an architecture in which the business logic of the application is organized in modules (services). Each module is a discrete service, and its internal design is independent of the nature and purpose of the requester (i.e., loosely coupled).

The Web has demonstrated the effectiveness of a loosely coupled architecture for improved interoperability across diverse platforms. IT applications are often large monolithic structures, “one-off” designs that meet specific sets of requirements. The size and one-off nature of such systems can lead to higher costs, longer development times, and difficulties in modifying such systems to reflect changes in business processes. In an SOA, independent business components are built that can be easily composed and possibly even automatically assembled into a system to support a work process. Application development shifts from a bottom-up approach that starts with the technical specifications to more of a top-down composition of existing processes.

A skeptic might have noted the similarity of ideal rendition of SOA vision with the Lego-block analogy for component assembly. Grady Booch, in a November 15, 2004 entry in his Web log, raises some of the concerns [Booch 05].

> Service-oriented architectures (SOA) are on the mind of all such enterprises - and rightly so - for services do offer a mechanism for transcending the multiplatform, multilingual, multisemantic underpinnings of most enterprises, which typically have grown organically and opportunistically over the years. That being said, I need to voice the dark side of SOA, the same things I've told these and other customers. First, services are just a mechanism, a specific mechanism for allowing communication across standard Web protocols. As such, the best service-oriented architectures

> seem to come from good component-oriented architectures, meaning that the mere imposition of services does not an architecture make. Second, services are a useful but insufficient mechanism for interconnection among systems of systems. It's a gross simplification, but services are most applicable to large grained/low frequency interactions, and one typically needs other mechanisms for fine-grained/high frequency flows. It's also the case that many legacy - sorry, heritage - systems are not already Web-centric, and thus using a services mechanism which assumes Web-centric transport introduces an impedence mismatch. Third, simply defining services is only one part of establishing a unified architecture: one also needs shared semantics of messages and behavioral patterns for common synchronous and asynchronous messaging across services.
>
> In short, SOA is just one part of establishing an enterprise architecture, and those organizations who think that imposing an SOA alone will bring order out of chaos are sadly misguided. As I've said many times before and will say again, solid software engineering practices never go out of style (crisp abstractions, clear separation of concerns, balanced distribution of responsibilities) and while SOA supports such practices, SOA is not a sufficient architectural practice.

Aspects of SOA and Web Services should support the solid software engineering practices that Booch advocates. The interface for Web Services has evolved from initially using the Simple Object Access Protocol (SOAP) as a remote procedure call to a document-style interface. This more elaborate interface can incorporate policies, service contracts, authentication and authorization information, and data schema that can be validated before the request is accepted. Policies for the kind of data and operations that are allowed can be defined for the endpoints of the communications chain.

An SOA raises the level of abstraction for the developer with a focus on business functionality, but an SOA does not necessarily solve the component integration problems, particularly with the implementation of non-functional requirements. For example, one observer of current practice noted that the tendency in pilots is to build point-to-point Web Services. The services are directly connected and tightly coupled. The security, service levels, exceptions handling, and so on are built into the code. Once those choices are hardwired into the code, reuse is no easier than with current practice.

Effective reuse with SOAs requires defining a shared infrastructure and managing the variances in non-functional requirements when the same business functionality is used in multiple operating contexts. Authentication and authorization are examples of shared infrastructure services. Organizational or regulatory polices for privacy or Sarbanes-Oxley compliance must also be properly implemented across dozens or even hundreds of components. The infrastructure has to provide the mechanisms to support that. The infrastructure for identity management serves multiple applications and may require integration of user data from multiple locations with independent owners. An infrastructure typically supplies what is common. Multiple users of a business service do not necessarily share non-functional requirements. Which design mechanisms provide the necessary variations in the non-functional requirements? A specific usage may require strong authentication and authorization mechanisms or have a high risk for a denial-of-service attack.

Loose coupling does not guarantee the lack of mismatches. A loosely coupled system can still generate contention problems with respect to resources shared among services. Reliability may be degraded if fault management is inconsistent across the services. A loosely coupled architecture, particularly one that crosses organizational boundaries, implies limited or no visibility and control of the called service. An in-depth analysis of the dependencies of the assembled system is very difficult, and may be impossible for some distributed systems. From one perspective, that lack of detailed analysis may be a benefit. As systems increase in size and complexity, such analysis is always incomplete, and any completed analysis only reinforces what may be undeserved confidence in the design. Applications that are forced to work in the absence of any presumed control or centralized authority and hence must behave more autonomously have the potential to be more robust than a more “trusting” design built on the assumptions derived from incomplete analysis.

The discussion of SOA raised concerns about the impact of loose coupling and treating a service as a black box. However, the application still controls the overall computational process and could use the state information it maintains to resolve faults raised by the service calls. An application with an SOA that used synchronous communications for services could use essentially the same program control structures for distributed services as those used for local function calls.

Instead of an application-based architecture, many business-to-business e-commerce systems currently use an event-driven architecture and asynchronous communications to avoid the overhead of having critical computing resources wait for the completion of an external service. A submitted business transaction represents an event to the receiving organization. Acknowledgements and other responses create events for the originator of the transaction. The submission of the transaction passes full control to the receiving system. The transaction status that might have been internally maintained by the application is now represented by persistent data that is accessed and updated as events are processed. The few controls and limited transactional state knowledge provided by an event-driven architecture limit the mechanisms that can be used to support software assurance.

In addition, some of the architecture principles applicable to an application, such as always mediating access to data, do not apply to an event-driven architecture. In an IT application, authentication and authorization can be reviewed whenever a data item is accessed. For an event-driven architecture, the authentication and authorization information is embedded in the interface document that describes the purchase order, and access controls could be implemented by one of the Web Services protocols. Careful encoding and design of the business order interface is required to satisfy a submitter's access and authentication requirements, as the submitter cannot directly mediate access to the data submitted in that transaction.

Model-Driven Development

The initial analysis in this note has concentrated on the problem of integrating existing components into a system. Such components meet both functional and non-functional requirements. An architecture design reflects the priority given to specific non-functional requirements. A real-time system will give priority to choices that improve performance and may use satisfactory and non-optimal techniques to meet modifiability requirements. Component differences in such tradeoff analyses are likely to create architectural mismatches.

In an SOA, the developer focus is on the business logic, which is then incorporated into a specific application architecture that is designed to meet a widely shared set of business integration requirements. Model-driven development generalizes that strategy by using models to separate the functional specifications from the implementation details. Whereas an after-development assessment of an application often has to generate a model from the source code and other design artifacts to support the analysis, model-based development proposes to generate code from well-specified models. Model-based development has been applied in a variety of domains and is likely to be more widely used. The Model-Driven Architecture (MDA) created by the Object Management Group (OMG)[95] defines a Platform-Independent Module (PIM) to capture the implementation-independent functional model and uses what OMG calls a model compiler to transform the PIM to a Platform-Specific Module (PSM) that would be deployed on a J2EE-based platform.

Microsoft's concept of a Software Factory attempts to integrate a model-based approach into more aspects of the development life cycle than MDA [Greenfield 04]. Again, the objective is to capture the knowledge about a specific application domain in a form that can be used to reduce the development effort required and yet improve the quality of that class of application. For example, for the SOA, a designer using the Microsoft approach might model the business processes as well as the user processes, i.e., common patterns of user interactions. Those models would be expressed in a domain specific language (DSL). Those usage models would influence deployment requirements for the Web Services,

[95] http://www.omg.com

which would have its own model. While the ever-increasing number of models might seem overwhelming, the complexity generated by the connections among business processes, user actions, and the system communication structures exists whether we formally represent it or not. The ad hoc management of those dependencies certainly increases the probability of mismatches as a system is assembled.

III. Architecture Integration Mechanisms

The software architecture is a key component of a software assurance case, but it can also be a source of problems. The design of the software architecture includes making the tradeoffs among the non-functional requirements. The preferred performance tactic may adversely affect operations or complicate maintenance. (See Architecture Risk Assessment content.) A method for implementing easy extensibility of a component may adversely affect security.

Postpone Architecture Decisions

Architectural mismatches may arise if a design decision is made too early in the development process. The details of design may depend on the choice of platform, such as J2EE or Microsoft's .NET. Other design choices may depend on the hardware or networks available on a host. One strategy is to delay the binding of a decision. Instead of incorporating an implementation choice into the source code, it may be possible to bind that choice at link-time by the selection of libraries, or bind it at system startup by the use of a configuration file The J2EE deployment description provides a way to manage deployment variances.

The Web Services protocols support the runtime specification for some aspects of the interface. A query to a discovery service can provide a description of a service and the necessary details of the interface. Information on work processes or security policy may be exchanged. However, the use of the Web interfaces introduces new vulnerabilities and hence new risks. Poorly constrained interfaces such as those that do not validate user input can be exploited by an attacker. The XML data stream generated by a Web Services interface is far more complex than the simpler procedural interfaces that generated buffer overflow vulnerabilities, That data stream could be subject to threats with respect to data encoding, by malformed XML data, or by changes in content that affect the operations performed. The validation required will be complex, but the standardization of Web Services also means that solutions can be shared and likely are incorporated into commercially available products.

Extensibility

One of the comments in the *Risk Digest* discussion of software assembly noted that commercial software products often have to be used without adaptation and hence could be used in a context for which they were not designed. Hence, software extensibility has become an increasingly important product attribute with more major industry players incorporating extensibility mechanisms such as plug-ins in their product lines. One strategy for software adaptation is illustrated by the Mozilla Firefox browser. Firefox implements the core functionality and then lets the user add additional functions by the use of extensions, which can be developed independently of Mozilla. The average Firefox user might install a few extensions; it would not be surprising if most users install ten or fewer extensions. The open source Eclipse Platform is designed to integrate software development tools. Everything for Eclipse is a plug-in. The initial estimate was that a large system might have several hundred plug-ins, but the plug-in count for several enterprise-class products built on Eclipse have passed the one thousand mark.

The use of plug-ins and other extensibility mechanisms certainly increases the difficulty of the software assurance task [Hoglund 04]. Whereas demonstrating the software assurance of a statically assembled system was a one-time event, assurance is an ongoing activity for an extensible architecture. There may be multiple sources for plug-ins, and a specific combination may not have been tested together. Security not only has to deal with the quality of the software imported from third-party sites but also with how that software is installed. There may be dependencies among plug-ins that create version control problems when those components are updated.

Finally, mobile code is increasingly used to extend the functionality of a system or to address changes in requirements or configurations. Mobile code means that the executable code, often written in a scripting language, is transmitted to a site for execution. Mobile code compounds the integration problems already noted for plug-ins. The increased power that mobile code provides the developer for rapidly changing system behavior also makes mobile code a very desirable mechanism for an attacker to exploit [McGraw 99].

IV. System Behavior: Component Interactions

> Software plays a major role in achieving the trustworthiness of an NIS, because it is software that integrates and customizes general-purpose components for some task at hand. In fact, the role of software in an NIS is typically so pervasive that the responsibilities of a software engineer differ little from those of a systems engineer. NIS software developers must therefore possess a systems viewpoint, and systems engineers must be intimately familiar with the strengths (and, more importantly, the limitations) of software technology [Schneider 99].

The analysis so far has considered the local behavior of a system: the misuse of a specific interface or problems caused by particular component design choices. That analysis starts at a detailed level, but as systems scale, the complexity of piecing together the details of local analysis is overwhelming. Many of the effective techniques for developing small systems do not scale. This section takes a top-down approach to systems software development.

Before we discuss integration and assembly from a systems perspective, we need to expand the boundaries of what composes a system. System behavior depends on the collective behavior of the computational components, the end users, and those responsible for administration and operation of the system. While a popular principle is to keep it simple, distributed software-intensive systems are complex. Often a decision that simplifies one aspect of the design does not reduce the overall complexity but simply transfers the complexity to another aspect of the system. (See Principles content.)

Operational errors are a frequent source of system failures as noted in *Trust in Cyberspace.*

> Errors made in the operation of a system also can lead to system-wide disruption. NISs are complex, and human operators err: an operator installing a corrupted top-level domain name server (DNS) database at Network Solutions effectively wiped out access to roughly a million sites on the Internet in July 1997 [Wayner 97]; an employee's uploading of an incorrect set of translations into a Signaling System 7 (SS7) processor led to a 90-minute network outage for AT&T toll-free telephone service in September 1997 [Perillo 97]. Automating the human operator's job is not necessarily a solution, for it simply exchanges one vulnerability (human operator error) for another (design and implementation errors in the control automation) [Schneider 99].

Techniques that simplified development or computational system components may adversely affect operations or ongoing system administration. The analysis of safety-related accidents often blames operators for the failure without asking if there were system characteristics that might have encouraged such an error. Careless management of system configuration files is a source of errors. The existence of multiple options to change system behavior by changes in configurations files or with extensions applied by users or administrators may simplify the integration task from the designer’s perspective but may generate errors by operations or the end user.

Systems Folklore

While the problems associated with networked computing systems may be receiving the most attention, the difficulties associated with managing large systems have a long history. John Gall first published *Systemantics* in 1975. The third edition was published in 2002 and renamed *The Systems Bible* [Gall 02]. While the wry humor in *The Systems Bible* is entertaining, the discussions should be a reminder that many aspects of large system assembly and integration are not well understood and that the methods that were successful for components and relatively simple systems do not necessarily scale to large systems.

In-depth safety analysis of accidents associated with large systems supports Gall's light-hearted exposition. For example, the thesis of *Normal Accidents* is that accidents are a normal event in complex high-technology systems [Perrow 99]. Unfortunately, some of those failures will be a surprise, as was demonstrated, for example, in the Three Mile Island incident. The responders to that incident were confronted with a system state that had not been anticipated by the designers and that provided evidence for Gall's axiom that "When a fail-safe system fails, it fails by failing to fail safely."

Grady Booch listed some of his favorite Gall axioms in his Web log on Feb 15, 2005 [Booch 05].

- A large system that is produced by expanding the dimensions of a smaller system does not behave like the smaller system.
- A complex system that works is invariably found to have evolved from a simple system that worked.
- A complex system design from scratch never works and cannot be made to work. You have to start over, beginning with a working simple system.
- Bad design can rarely be overcome by more design, whether good or bad.
- Loose systems last longer and function better.

Some additional items applicable to the discussion in this report include

- Any large system is going to be operating most of the time in failure mode.
- Reality is more complex than it seems.
- The mode of failure of a complex system cannot ordinarily be determined from its structure.
- One does not know all the expected effects of known bugs.

Gall's axioms are applicable to human-intensive organizational systems and most ring true for computing systems. Leveson has been studying the safety failures associated with software-intensive systems. Such systems are relative newcomers to the systems world. She argues for a system-theoretic approach to safety in that context, and a number of her comments are applicable to software properties other than safety.

System Models

The error analysis for integration categories I and II was bottom up and similar to what is called event-chaining in safety engineering [Leveson 95]. The assumption for almost all causal analysis for engineered systems today is a model of accidents that assumes they result from a chain (or tree) of failure events and human errors. From an observed error, the analysis backward-chains and eventually stops at an event that is designated as the cause. Usually a root cause selected from the chain of events has one or more of the following characteristics: (1) it represents a type of event that is familiar and thus easily acceptable as an explanation for the accident, (2) it is a deviation from a standard, (3) it is the first event in the backward chain for which a "cure" is known, and (4) it is politically acceptable as the identified cause [Leveson 05].

> Event-based models of accidents, with their relatively simple cause-effect links, were created in an era of mechanical systems and then adapted for electro-mechanical systems. The use of software in engineered systems has removed many of the physical constraints that limit complexity and has allowed engineers to incorporate greatly increased complexity and coupling in systems containing large numbers of dynamically interacting components. In the simpler systems of the past, where all the interactions between components could be predicted and handled, component failure was the primary cause of accidents. In today's complex systems, made possible by the use of software, this is no longer the case. The same applies to security and other system properties: While some vulnerabilities may be related to a single component only, a more interesting class of vulnerability emerges in the interactions among multiple system components. Vulnerabilities of this type are system vulnerabilities and are much more difficult to locate and predict [Leveson 05].

Distributed decision making across both physical and organizational boundaries is a necessity for software-intensive, complex, human-machine systems. Security provides a simple example. For a good many years, a firewall combined with access control for the network provided reasonable security, but with the complexity of the multiple connections required for increased integration of business systems, that single perimeter has been replaced by security zones, that is, by multiple perimeters. As work processes extend beyond the corporate IT perimeter and incorporate services and data provided by external systems, the concept of a perimeter becomes even more elusive. It would not be surprising that security zones become a zone of one as each connection is individually evaluated, and that evaluation reflects the dynamically changing assurance associated with that link. The central control represented by a firewall-protected perimeter has been replaced by multiple independent control points, each of which may have only limited knowledge of the behavior of the remote services. The limited available knowledge is partially a consequence of using loosely coupling services to avoid dependencies on implementation features.

Emergent Properties

Under some assumptions, a computing system can be decomposed in much the same manner as a physical system so that the parts can be examined separately and the system behavior can be predicted from the behavior of components. The validity of the analysis assumes that

- the components or events are not subject to feedback loops and non-linear interactions
- the behavior of the components is the same when examined alone as when they are playing their part in the whole
- the principles governing the assembly of the components into the whole are straightforward, that is, the interactions among the subsystems are simple enough that they can be considered separate from the behavior of the subsystems themselves [Leveson 05, Weinberg 75]

These assumptions fail quickly for a complex software-intensive system. A systems approach as proposed by Leveson for safety focuses on systems taken as a whole, not on the parts examined separately. Key concepts in basic system theory are emergence and hierarchy. A system is described in terms of a hierarchy of levels. Each level is characterized by having emergent properties. Those emergent properties do not exist at the lower levels.

> Safety is an emergent property of systems. Determining whether a plant is acceptably safe is not possible by examining a single valve in the plant. In fact, statements about the "safety of the valve," without information about the context in which that valve is used, are meaningless. Conclusions can be reached, however, about the reliability of the valve, where reliability is defined as "the ability of a system or component to perform its required functions under stated conditions for a specified period of time," i.e., that the behavior of the valve will satisfy its

> specification over time and under given conditions. This is one of the basic distinctions between safety and reliability: Safety can only be determined by the relationship between the valve and the other plant components—that is, in the context of the whole. Therefore it is not possible to take a single system component, like a software module, in isolation and assess its safety. A component that is perfectly safe in one system may not be when used in another. Attempts to assign safety levels to software components in isolation from a particular use, as is currently the approach in some international safety standards, is misguided [Leveson 05].

Emergent properties are analogous to the properties of biological and social systems, where each actor performs simple local behaviors involving interactions with other actors but without complete knowledge of who else is participating. Popular examples include behavior in ant colonies or in flights of geese. A social system example is the use of chat rooms. The management of the Internet is an example of an emergent behavior for a distributed computing system.

Leveson noted that vulnerabilities generated by interactions among multiple system components are much more difficult to locate and predict than the vulnerabilities associated with a specific component or technology. Unfortunately, the constructive side of a systems approach is equally difficult. The system design challenge is to specify the appropriate local rules of behavior for components that generate the desired emergent behavior for the system or community, or at least identify local behavior that may lead to undesirable global behavior. The system designer is caught between the proverbial rock and a hard place. For complex systems, a bottom-up approach to design does not scale well, while a top-down approach faces the challenge of predicting system behavior from local interactions. Gall's observation that a large system that is produced by expanding the dimensions of a smaller system does not behave like the smaller system might be explained by the need to incorporate emergent properties for large systems [Forrest 90]. At least one large financial organization is using emergent concepts to help manage system complexity with respect to security.

General Advice

Defects in some form are a given for a large system. A persistent theme has been the increased likelihood of errors as the boundary for distributed computation expands and the control and visibility for the remote components used in those computations diminishes. There are constraints that are derived from schedule and budget that limit the freedom of selecting the system components and in choosing a development process. An ATM machine or a medical system used for patient care might be a general purpose PC. General-purpose software such as COTS products will be used in security-sensitive systems.

Many existing systems were developed as self-contained or closed systems that had significant control over their operating environment. The design of those systems might impose operational constraints in terms of permitted access or computing systems used that enabled the system architects to create homogeneous operational zones or compartments that simplified development. On the other hand, corporate IT systems are losing the distinctions that help define compartments with the blending of internal and external networks (perimeter controls), corporate and personnel computing equipment (desired behavior), personnel and corporate information (rights management), and personnel and corporate accountability (regulations and corporate policy).

Trust in Cyberspace suggests a pragmatic strategy to deal with the realities of system development [Schneider 99].

> Fortunately, success in building an NIS does not depend on writing software that is completely free of defects. Systems can be designed so that only certain core functionality must be defect free; defects in other parts of the system, though perhaps annoying, become tolerable because their impact is limited by the defect-free core functionality. It now is feasible to contemplate a system having millions of lines of source code and embracing COTS and legacy components,

> since only a fraction of the code actually need be defect free. Of course, that approach to design does depend on being able to determine or control how the effects of defects propagate. Various approaches to software design can be seen as providing artillery for attacking the problem, but none has proved a panacea. There is still no substitute for talented and experienced designers.

“Engineering for Failure,” the title of an interview with Bruce Lindsay on reliability and error management for database technology, captures a key point of the recommendation [Lindsay 04].

Such a strategy could include the development of misuse cases as part of requirements elicitation. Those misuse cases are likely to be general and will not document what might go wrong with specific system interfaces. For security, the advice is “to think like an attacker,” but a system designer may not be knowledgeable of the techniques that might exploit an interface.

Documenting the dependencies among the assembled components is an essential step in understanding possible system failures and identifying components impacted by defects. Such analysis might identify potential single points of failure or possible contention for shared resources. Dependency analysis should be part of an architectural risk analysis. The dependency analysis for legacy or COTS components is required for managing any faults generated by those components. The dependency analysis might also suggest that what was thought of as an elegant and simple top-level design is actually chaotic below the surface.

Successful integration depends on solid software engineering practices. Software development involves a series of decisions, including many tradeoffs. Some options may be obvious. A new design is likely to follow the principle to use the least privileges necessary for the task rather than using an account with full system privileges. However, with the increased deployment of networked-integrated systems, vulnerabilities are equally likely to arise from how components are composed and then used to support the business processes. The system design needs to consider such faults.

Failure states that represent emergent behavior rather than component failure are difficult to recognize and mitigate. Such deviant emergent behavior may be a consequence of local component actions on error recovery and propagation. Most errors start as a component event: a failed calculation, an error returned by a function call, or a non-response from a remote service. For some defects, recovery can be contained within the component, but in other instances, the failure has to be reported and other components have to respond to that defect and the recovery actions taken.

Leveson separates safety accidents into two types: those caused by failures of individual components and those caused by dysfunctional interactions between non-failed components. In most software-related accidents, the software operates exactly as specified, that is, the software, following its requirements, commands component behavior that violates system safety constraints, or the software design contributes to unsafe behavior by human operators (i.e., system-level analysis identifies multiple contributing factors rather than a single failure) [Leveson 05].

Often the desired objective following a successful security attack is to identify a root cause such as a buffer overflow and to describe the event in terms of a linear sequence of attacker actions. Leveson argues that with respect to safety the traditional event-chain model, with its emphasis on component failure, is inappropriate for today’s software-intensive, complex, human-machine systems with distributed decision making across both physical and organizational boundaries. Leveson’s argument may be applicable to software assurance for complex systems. Attacker tactics evolve in response to changes in defensive strategies, system usage, and technologies deployed, and future tactics might exploit system interactions rather than just component vulnerabilities. In practice, only a subset of potential risks can be considered based on time and resources. Selecting the appropriate subset is a key challenge. There is also the issue of considering highly unlikely events—many potential failures are not even analyzed because they are considered too remote, but the need to consider high impact events remains.

References

[Anderson 01] Anderson, Ross. *Security Engineering: A Guide to Building Dependable Distributed Systems.* New York, NY: John Wiley & Sons, 2001.

[Blakley 04] Blakley, Bob & Heath, Craig. *Security Design Patterns.* The Open Group, 2004. http://www.opengroup.org/bookstore/catalog/g031.htm (2005).

[Booch 05] Booch, Grady. *Architecture Web Log*. http://www.booch.com/architecture/blog.jsp (2005).

[Ellison 05] Ellison, Robert J. *Trustworthy Composition: The Challenges for the Practitioner* (CMU/SEI-2005-TN-026). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2005.

[Forrest 90] Forrest, Stephanie, ed. *Emergent Computation: Self-Organizing, Collective and Cooperative Phenomena in Natural and Artificial Computing Networks.* Cambridge, MA: MIT Press, 1991. Also published *as Physica D* special issue 42, 1-3 (1990*).*

[Gall 02] Gall, John. *The Systems Bible*. Walker, MN: The General Systemmantics Press, 2002.

[Garlan 94] Garland, David; Allen, Robert; & Ockerbloom, John. "Architectural Mismatch: Why Re-use Is So Hard." *IEEE Software,* (November 1994): 17-26.

[Greenfield 04] Greenfield, Jack & Short, Keith. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Indianapolis, IN: Wiley Publishing, 2004. Microsoft Software Factory link: http://lab.msdn.microsoft.com/teamsystem/workshop/sf/default.a

[Hoglund 04] Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code.* Boston, MA: Addison-Wesley, 2004.

[Lampson 03] Lampson, B. W. "Software Components: Only the Giants Survive," 113–120. *Computer Systems: Papers for Roger Needham, K. Spark-Jones and A. Herbert (editors)*. Cambridge, U.K.: Microsoft Research, February 2003.

[Leveson 95] Leveson, Nancy G.. *Safeware: System Safety and Computers.* Reading, MA: Addison-Wesley, 1995.

[Leveson 05] Leveson, Nancy. "A Systems-Theoretic Approach to Safety in Software-Intensive Systems.” *IEEE Transactions on Dependable and Secure Computing 1,* 1 (January-March 2004): 66-86.

[Lindsay 04] “Engineering for Failure.” *ACM Queue 2,* 8 (November 2004). http://www.acmqueue.org/modules.php?name=Content&pa=sho

[McGraw 99] McGraw, Gary & Felton, Edward W. *Securing Java: Getting Down to Business with Mobile Code, 2nd Edition.* New York, NY: John Wiley & Sons, 1999.

[McIlroy 68] McIlroy, M. D. “Mass Produced Software Components, 138-151. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE*. Garmisch, Germany, Oct. 7-11, 1968. Brussels, Belgium: Scientific Affairs Division, NATO, 1969. http://homepages.cs.ncl.ac.uk/brian.randell/NATO.

[Meyer 92] Meyer, Bertrand. “Design by Contract.” *IEEE Computer 25*, 10 (October 1992): 40-51.

[Neumann 95] Neumann, Peter G. *Architectures and Formal Representations for Secure Systems* (CSL Report 96-05, Final Report, Project 6401). Menlo Park, CA: SRI International, 1995.

[Neumann 04] Neumann, Peter G. *Principled Assuredly Trustworthy Composable Architectures* (Final Report to DARPA, CDRL A001). Menlo Park, CA: Computer Science Laboratory, SRI International, December, 28, 2004. http://www.csl.sri.com/users/neumann/chats4.html.

[Perillo 97] Perillo, Robert J. “AT&T Database Glitch Caused ‘800’ Phone Outage.” *Telecom Digest 17*, 253 (September 18, 1997). http://massis.lcs.mit.edu/archives/back.issues/1997.volume.17/vol17.iss251-300.

[Perrow 99] Perrow, Charles. Normal Accidents: *Living with High Risk Technologies*. Princeton, NJ: Princeton University Press, 1999.

[Risks Digest 05] Initial Comment: Paul Robinson, Risks Digest, Number 23.73. http://groups-beta.google.com/group/comp.risks/browse_thread/t (2005).

Responses are in 23.74, 23.75, 23.76, and 23.77. Each digest includes a link to the following and previous issues.

[Saltzer 75] Saltzer, Jerome H. & Schroeder, Michael D. “The Protection of Information in Computer Systems.”

	Proceedings of the IEEE 63, 9 (September 1975): 1278-1308.
[Schneider 99]	Schneider, Fred B., ed. *Trust in Cyberspace*. Washington, DC: National Academy Press, 1999. http://www.nap.edu/books/0309065585/html/R1.html.
[Towson 97]	Towson, Peter. “AT&T Database Glitch Caused ‘800’ Phone Outage.” *Telecom Digest 17*, 253 (September 18, 1997). http://hyperarchive.los.mit.edu/telecom-archives/.
[Wayner 97]	Wayner, Peter. “Human Error Cripples the Internet.” *New York Times*, July 17, 1997. http://www.nytimes.com/library/cyber/week/071797dns.html.
[Weinberg 75]	Weinberg, Gerald. *An Introduction to General Systems Thinking*. New York, NY: John Wiley & Sons, 1975.

SEI Copyright

Velden

Naam	Waarde
Copyright Holder	SEI

Velden

Naam	Waarde
is-content-area-overview	false
Content Areas	Best Practices/Assembly, Integration, & Evolution

[1] http://www.sei.cmu.edu/about/legal-permissions.html

SDLC Relevance	Cross-Cutting
Workflow State	Publishable